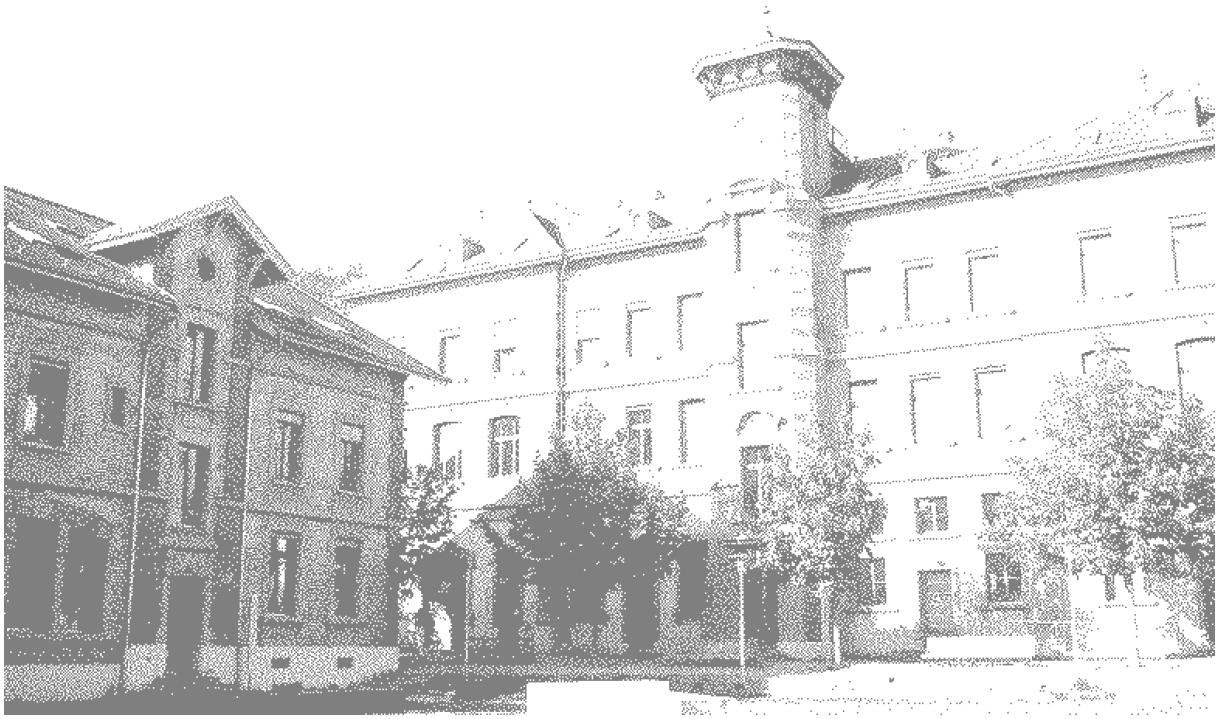
UNIVERSITÄT
BERN

Persistent Caching in Information-Centric Networking

C. Anastasiades, A. Gomes, R. Gadow, T. Braun

Technischer Bericht IAM-15-001 vom 22. Mai 2015

Institut für Informatik und angewandte Mathematik, www.iam.unibe.ch



Persistent Caching in Information-Centric Networking

Carlos Anastasiades, Andre Gomes, Rene Gadow, Torsten Braun

Technischer Bericht IAM-15-001 vom 22. Mai 2015

CR Categories and Subject Descriptors:
C.2.1 [Computer-Communication Networks]: Network Architecture and Design; C.4 [Computer Systems Organization]: Performance of Systems - Design studies

General Terms:
Design, Measurement, Performance

Additional Key Words:
Information-centric networks, persistent caching, CCN, repository

Institut für Informatik und angewandte Mathematik, Universität Bern

Abstract

Information-centric networking (ICN) is a new communication paradigm that aims at increasing security and efficiency of content delivery in communication networks. In recent years, many research efforts in ICN have focused on caching strategies to reduce traffic and increase overall performance by decreasing download times. Since caches need to operate at line-speed, they have only a limited size and content can only be stored for a short time. However, if content needs to be available for a longer time, e.g., for delay-tolerant networking or to provide high content availability similar to content delivery networks (CDNs), persistent caching is required. We base our work on the Content-Centric Networking (CCN) architecture and investigate persistent caching by extending the current repository implementation in CCNx. We show by extensive evaluations in a YouTube and webserver traffic scenario that repositories can be efficiently used to increase content availability by significantly increasing the cache hit rates.

Contents

1 Introduction

Information-Centric Networking (ICN) has been proposed to address shortcomings of the Internet Protocol, such as scalability for increasing mobile data traffic [1] and security [2]. ICN messages are routed based on names instead of endpoint identifiers. Content is identified by unique names, which enable concurrent streams to be aggregated and content to be cached in any node. Because content is signed, integrity and authenticity of retrieved content is ensured and it is not important which node provided the content copy.

In recent years, extensive research efforts, e.g., [3], [4], [5], [6], [7], [8], [9], have been performed to address caching in wired information-centric communication for the Future Internet. The basic idea of caching in ICN is to keep received data in buffers to satisfy similar requests. The cache is considered as short-term storage to avoid retransmissions over the entire path to a content source in case of collisions or synchronize multiple concurrent requesters of the same content. In the latter case, caches can consolidate even slightly time shifted requests, depending on how long content is cached, to reduce network traffic.

With the vast profileration of mobile devices in recent years, mobile data traffic has increased drastically and is expected to increase even more in the following years. According to Cisco's Global Mobile Data Traffic Forecast report [10], 4G will be more than half of the total mobile traffic by 2017 and the average traffic amount per smartphone will increase fivefold by 2019. To reduce traffic and increase performance, ICN caching can be integrated into LTE mobile networks [11]. However, caches need to operate at line-speed, thus, current memory technologies impose limitations. Fast memory is expensive, power hungry and only available in small capacities [12]. Furthermore, caches are implemented in volatile storage, which is cleared, i.e., data loss, in case of power outages.

Therefore, in some scenarios, short-term caching may not be enough and content needs to be persistently stored (at the expense of slightly slower access times). This is required, e.g., for delay-tolerant networking [13], [14], custodian-based information sharing [15] or to enable high availability and performance similar to content distribution networks (CDNs) by dynamically storing content in regions of high demand.

In this work, we investigate persistent caching for content distribution. Our work is orthogonal to existing ICN research on caching, because it can be combined with fast (short-term) caching. While received and forwarded content will automatically be stored in the cache for a short time, persistent

storage can be used to store only a subset of it for a longer time. For example, real-time audio streams from phone conferences may be stored in the cache, but it may not be required to keep them for a long time. In contrast, large static files, such as multimedia files or pictures may be valid for a longer time and can be cached at persistent storage closer to requesters.

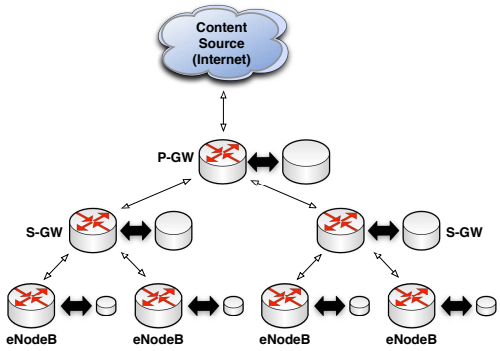


Figure 1: Hierarchical Caching in LTE Network.

We envision to extend hierarchical networks such as LTE mobile networks by adding persistent caches as shown in Figure 1. Traffic from users is forwarded in a hierarchical way from evolved Node Bs (eNBs) to Serving Gateways (S-GWs) and from there to a PDN gateway (P-GW). The P-GW provides connectivity to external networks such as the Internet. Persistent storage may be deployed alongside content routers attached to eNBs, S-GWs and P-GWs, and stores a subset of content forwarded through these routers. This enables storing very popular content of the day, e.g., electronic newspapers or popular videos, at the edge of the network to improve network performance. It also means that many requests of popular content may be satisfied already at edge routers, while requests for less popular content may be forwarded further to the next content router, which may hold a cached copy of the content. Therefore, only unpopular content, for which caching would not yield any benefits, would need to be retrieved all the way from the content source. Such an approach brings multiple advantages from the perspective of both end users and Mobile Network Operators (MNOs). For the former, perceived performance significantly increases due to lower content access latency, either delivered directly by caching or by one of its side effects, i.e. backhaul traffic experiences a major reduction, allowing faster content downloads from more distant sources. For the latter, Operational Expenditures (OPEX) can be reduced up to 36% [16] due to the lower load of the network infrastructure.

We base our investigations on Content-Centric Networking (CCN) [17], which is a popular ICN architecture. Persistent storage in CCN is provided by repositories. For this work, we extended the repository implementation in CCNx, the open source reference implementation of CCN, to support persistent caching and have evaluated its feasibility by extensive tests using various request models.
The remainder of this paper is organized as follows. In Section 2, we give an overview on CCN and relevant work on caching. Our design for persistent caching is described in Section 3. Evaluation results are presented in Section 4 and discussed in Section 5. Finally, in Section 6, we conclude our work and give an outline for future work.

2 Content-Centric Networking

2.1 CCN Concepts and Implementation

Content-Centric Networking (CCN) is based on two messages: *Interests* to request content and *Data* to deliver content. Files are composed of multiple segments, which are included in a Data message, and users need to express Interests in every segment to retrieve a complete file. CCNx [18] provides an open source reference implementation of CCN. The core element of CCNx is the CCN daemon (CCND), which performs message processing and forwarding decisions. Links from the CCND to applications or other hosts are called *faces*.
The CCND has three main memory components: the Content Store (CS), the Pending Interest Table (PIT) and the Forwarding Information Base (FIB). When an Interest is processed, the CS, which is used as a cache, is checked to verify whether the content is available locally. If not, the PIT, which stores forwarded Interests in a soft state so that Data messages can travel the same path back, is checked to verify if the content was already requested. If it was not, the FIB is used to forward the Interest as it contains forwarding entries to direct it towards potential content sources. Additionally, every received and forwarded Data message is temporarily stored in the CS and Least Recently Used (LRU) replacement strategies are applied for cache replacement.
Content sources can publish and persistently store content in repositories. The content is stored in the wire-format, i.e., including headers and signatures, in a file, the *repofile*. For fast access to content in the repofile, the repository keeps references to content in a B-tree.
Currently, it is possible to publish new content in a repository with the *ccnputfile* application, retrieve content and store it in the repository with a *start-write Interest* or synchronize collections among repositories with the *Sync Protocol*. However, there is no way of deleting content from a repository besides resetting it, which results in the deletion of all stored content.

2.2 Caching

Caching in information-centric networking has been subject to extensive research in recent years. In CCN, content is cached everywhere along the downloading path resulting in high cache redundancy. It has been shown that popular content tends to be cached at the leafs of the network [3] and, therefore, allocating more storage resources to edge routers than core

routers is beneficial in terms of performance [4] and energy consumption [19]. To avoid redundant caching, several strategies have been proposed such as limiting the number of cached copies along the path to one [5], probabilistic caching based on distance from the content source [4], or apply network coding to ensure content diversity caches [8], [9]. Other approaches are based on coordination for content deletion, e.g., pushing deleted content one-level upstream the caching hierarchy [6] or adapting the number of cached chunks based on the file's popularity [7].
However, storage in the CS is limited and cached content is only available for a limited amount of time to support faster retransmissions in case of collisions and to synchronize multiple concurrent (or slightly time shifted) requests. If content should be available for a longer time, e.g., for delay-tolerant networking or to ensure high availability and performance similar to content distribution networks (CDNs), it should be persistently stored. In this work, we investigate persistent caching based on the current CCNx repository implementation. While every content in CCN needs to go through the CS, repositories can monitor the network traffic and store only a subset of content, e.g., non real-time data or large static files such as pictures or videos.

3 Design and Implementation of Persistent Caching

In CCN, persistent storage is provided by repositories. The current repository implementation in CCNx stores all content in the *repofile* and maintains references to the content in a B-tree.
Content sources publish content in repositories to make them available to other nodes. To use repositories for caching, content deletion needs to be introduced in an automatic way, e.g., based on popularity. However, we do not maintain popularity counters for two reasons. First, popularity counters would need aging mechanisms, introducing significant additional complexity. For example, content that has been requested extensively one year ago may be less popular in the near future than content that has been frequently requested in the last hours, although the absolute number of requests would be lower. We prefer simplicity over complex solutions to minimize the processing overhead in content routers. Second, since content is also cached in non-persistent memory, request statistics at repositories would still not reflect the effective number of requests by end-users. In addition, deletion operations should only be performed if free space is required and not based on aging-based timers because it may still be useful. Therefore, in this work, we maintain access information and delete content that has not been requested recently. There are two main differences to LRU strategies. First, deletion operations are performed based on content and not individual chunk popularities. Second, multiple content objects may be deleted at the same time to free space if a certain storage utilization threshold is reached because deletions in the filesystem take more time than in main memory.

3.1 Data Structures

Figure 2 illustrates data structures required to enable content deletion for persistent storage. The *delete_queue* maintains an element for every content object in the *repofile*. The basic idea is to move requested content to the tail (bottom) of the queue such that unpopular content can be found at the head (top) of the queue. Therefore, if content needs to be deleted, it can be removed from the head.
Figure 2 shows that the *delete_queue* is implemented as doubly linked list, on which every element has a pointer to the previous and next elements. In addition, every queue element has a pointer to another linked list of

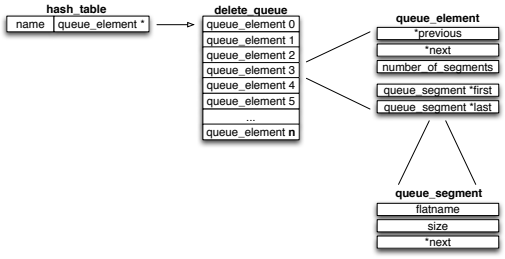


Figure 2: Additional data structures for persistent storage

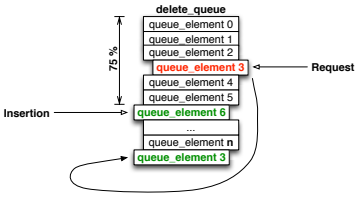
queue_segments, i.e., the individual segments of the content. Besides a pointer to the next element, we also maintain a pointer to the last segment in the list to avoid long list traversals when including segments of large content. The *queue_segment* contains the flat name of a segment to find the content (and its reference to the *repofile*) in the B-tree. When we need to find a *queue_element* quickly, we use a *hash table* to get its reference in the *delete_queue* based on a lookup of the base name, i.e., content name without segment numbers.
In contrast to related work on CCN caching, we keep content based on object granularity and do not make individual caching decisions for every segment/chunk. Because content in CCN is requested sequentially based on the pipeline size, high variability in chunk downloads would degrade overall download performance. More information on chunk-based vs. object-based caching can be found in Section 5.

3.2 Processing

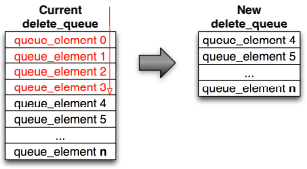
In this section, we describe processing procedures in the *delete_queue*.

3.2.1 Inclusion

Content information is stored based on object granularity. When a segment is received, the content name, i.e., base name without segment number, can be extracted. Based on a hash table lookup, *delete_queue* entries of existing content can be found quickly. In this case, only a new *queue_segment* needs to be added to the *delete_queue* entry. If it is new content, the content is included in the *delete_queue*. As Figure 3a shows, we include new entries in the middle of the lower half, i.e., at 75% of the queue. If content would be included in the upper 50% of the delete queue, new popular content could be deleted almost instantly, e.g., if the inclusion



(a) Insertion and Update



(b) Deletion

Figure 3: Delete Queue Processing.

is just before a content deletion, since up to 50% of the repofile is deleted during a deletion operation (see subsection 3.2.3). In addition, it is not appended to the tail such that unpopular content can quickly reach the head of the *delete_queue*, while popular content can go to the tail.

3.2.2 Queue Update

Figure 3a illustrates also update operations on the *delete_queue*. Every time content is requested, the corresponding element in the *delete_queue* is pushed to the tail of the queue. This push operation can be performed for every requested segment, every n-th segment or only the first segment. If every n-th segment is processed, there would be a tendency of larger files at the tail of the queue, since they have more segments and, thus, more pushing operations. Therefore, despite some disadvantages (see Section 5), we decided to consider only the first segment of a content object as trigger for pushing operations.

3.2.3 Deletion

A deletion operation is initiated, if the *repofile* has reached a certain size, i.e., the *repofile threshold*. Then, a deletion operation is performed by deleting a configurable percentage of the *repofile*, i.e., the *deletion ratio*. A deletion operation is initiated after a file inclusion, if the *repofile threshold* has been exceeded. Please note that in CCNx, file sizes are only known

when the last segment has been received with the *final bit* set. Therefore, the *repofile threshold* is a soft threshold and the *repofile* size can become slightly larger than the threshold depending on the size of included files, i.e., we do not perform deletion operations during file inclusions but rather afterwards.
Figure 3b shows modifications on the *delete_queue* due to deletion operations. A deletion operation is performed by the following steps.

1. Prevent the repository daemon from accepting new content while the deletion operation is being performed. If new content arrives during local deletion operations, it will only be stored temporarily in the content store. However, other repositories on the path to the requester will store it persistently.

2. Start at the head of the *delete_queue* and iterate through the elements until the *deletion ratio* is reached. All content entries up to this point will be deleted (red part in Figure 3b) and the lower part becomes the new *delete_queue*.

3. Every delete queue element contains multiple *queue_segments*. The *queue_segments* of all deleted content objects need to be sorted based on their position in the *repofile* such that every B-tree entry and the repofile need only to be processed once (see next step). In our current implementation, we use the $\mathcal{O}(n \log n)$ merge-sort algorithm for sorting.

4. All content from the *repofile* (except deleted segments) are copied to a new *repofile*. This is required because file systems do not support selective deletions inside files. Due to deletions, content is copied to other positions in the new *repofile*, thus, reference values in the B-tree need to be updated.

5. All B-tree entries of deleted content are removed.

6. Instruct the repository daemon to start accepting new content again.

To limit service interruptions from deletions, a (read-only) repository can be started to provide content from the old *repofile*. Otherwise, Interests may just be forwarded and satisfied by persistent caching at the next router level. Thus, only in the worst case Interests would be forwarded all the way to the content source.

4 Evaluation

Persistent caching has been implemented by extending the repository implementation in CCNx 0.8.2, and extensive evaluations have been performed in different scenarios on physical servers of a Linux cluster [20].

4.1 Scenarios

Figure 4 shows our evaluation topology. We evaluate the performance of an edge router, e.g., at an eNB, that continously receives requests from the network according to YouTube and webserver traffic models. The edge router is connected to a local repository, which is responsible for persistent caching. Independent of the network topology, an edge router has a *downstream face* from which file requests are received and content is returned and an *upstream face* where received Interests are forwarded and new content is received, i.e., file inclusions at the persistent cache of the edge router. The evaluation parameters are listed in Table 1.

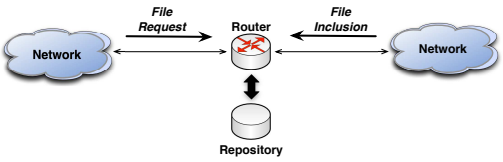


Figure 4: Network Scenario.

Similar to existing ICN literature [21], we assume that content popularity follows a Zipf distribution. We use 20 popularity classes and perform evaluations with parameters α set to 1 and 2. A parameter of $\alpha = 1$ is considered realistic for webserver traffic and $\alpha = 2$ is used for YouTube traffic [21]. Several studies have shown [22], [23] that most files are unpopular and only a few files are very popular. Therefore, we map the number of files in all popularity classes to a Zipf distribution $\alpha = 1$ with inverted classes, i.e., most files are included in class 19 and fewest files in class 0. The file sizes in each popularity class vary as well. Based on existing YouTube models [24], we set the file size distribution for our YouTube scenario to a gamma distribution with $\alpha = 1.8$ and $\beta = 5500$. Our YouTube files are between 500KB and 100MB, while most files are between 2 and 10MB (9.9MB mean). The file sizes for web server traffic are considerably smaller [25]. File sizes have increased in the last years and we believe that file sizes will increase even more in future information-centric

Parameter	**YouTube**	**Webserver**
Requests	every 5s	
Request Popularity	Zipf distribution with $\alpha = 2$ $\alpha = 1$	
File distribution per popularity class	Zipf distribution, $\alpha = 1$ mapped to inverse classes	
New Files	every 10s	
File sizes per popularity class	Gamma distribution, $\alpha = 1.8$, $\beta = 5500$ min. 500KB max. 100MB	Gamma distribution, $\alpha = 1.8$, $\beta = 1200$ min. 50KB max. 50MB
Repofile thresholds	2GB, 4GB, 8GB	8GB, 12GB, 16GB
Deletion ratios	50%, 25%	
Effective duration	86400s (1 day)	

Table 1: Evaluation parameters.

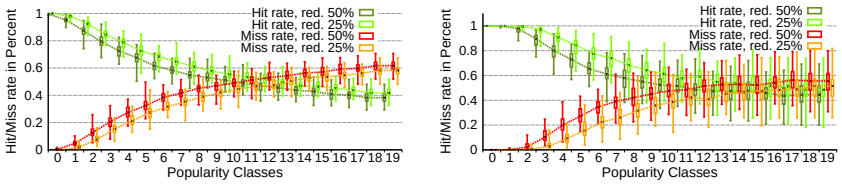
networks. Transmitted ICN packets need to have a certain minimum size to be efficient, e.g., segment size of 4096 bytes or more, to avoid too large overhead for content headers including names and signatures. Therefore, we believe that for future ICN traffic, many small files may be aggregated to larger data packets or ICN would only be applied to large static files, e.g., pictures or embedded videos, and not small text files that may change frequently. Therefore, we use a gamma distribution with $\alpha = 1.8$ and $\beta = 1200$ for the webserver scenario. Our webserver files are between 50KB and 50MB, however, most files are between 750KB and 1250KB (2MB mean).

In our scenarios, requests are performed periodically, i.e., 1 new content request every 5 seconds. The requested content from the popularity class (Zipf distribution) is selected randomly among all created content objects in that popularity class. To simulate a dynamically growing file catalog and to evaluate the performance of persistent caching with regular deletion operations, we create and request new files every 10s. These files are included into the repository, i.e., file inclusions, as mentioned above.

For every scenario, we evaluate various *repofile tresholds* and *deletion ratios* of 50% (DR50) and 25% (DR25) of the *repofile*. We measure the performance of persistent caching during operation, i.e., the repository is filled initially with content and we collect statistics after a first deletion operation has been performed. The effective evaluation starts after the first deletion and lasts 86400 seconds (1 day). Thus, in one day we create approximately 85.54 GB of data in the YouTube scenario and 18.67GB of

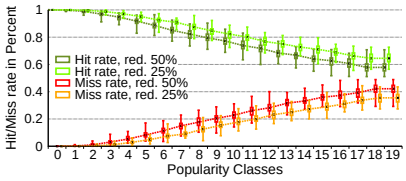
data in the webserver scenario. Every configuration has been evaluated and repeated 100 times on physical servers that run a CCN router with persistent caching.

4.2 Hit and Miss Rates

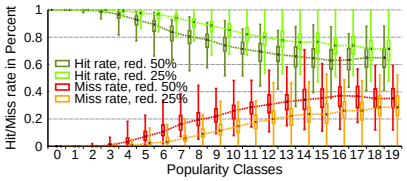


(a) webserver, repofile threshold 2GB

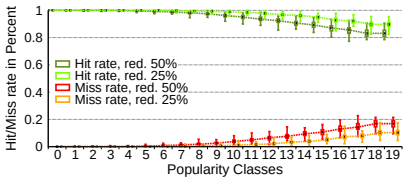
(b) YouTube, repofile threshold 8GB



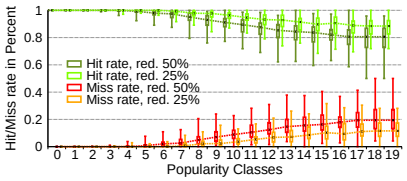
(c) webserver, repofile threshold 4GB



(d) YouTube, repofile threshold 12GB



(e) webserver, repofile threshold 8GB



(f) YouTube, repofile threshold 16GB

Figure 5: Hit and Miss rates for webserver and YouTube scenarios.

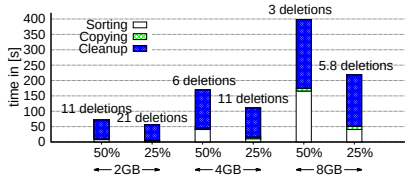
In this subsection, we evaluate the cache hit rates of our repository implementation in the YouTube and webserver scenario. Figure 5 shows the cache hit and miss rates for all popularity classes in different configurations. The y-axis shows the hit/miss rates and the x-axis indicates the popularity class. The figures on the left side are obtained for our webserver scenario, i.e., requests with Zipf distribution $alpha = 1$, and the figures on the right side show the YouTube scenario with Zipf distribution $alpha = 2$. Figure 5a shows the hit and miss rates in the webserver scenario with a repofile threshold of 2GB. The dark green boxplots show the hit rates for

a deletion ratio of 50% (DR50) and the light green boxplots for a deletion ratio of 25% (DR25). The overall hit rate of DR25 is slightly higher, i.e., 81%, compared to 77.5% with DR50. For high popularity classes, such as classes 0 and 1, the difference between DR50 and DR25 is smaller because files from these classes are barely deleted in both cases. However, for classes 3-17, the difference between DR25 and DR50 is larger by up to 6.3% because these files are kept more likely with DR25, while they are deleted with DR50. The red boxplots show the miss rates for DR50 and the orange boxplots for DR25. For DR50, hit rates are higher than miss rates up to files from popularity class 11, while for DR25, hit rates are better for more files, i.e., up to class 13. Even for the most unpopular content in class 19, DR50 results in a slightly higher miss rate of 61.5% compared to 58.1% with DR25. Therefore, freeing space too aggressively, i.e., DR50, has a noticeable impact on cache hit rates in the webserver scenario.
Figure 5b shows the hit and miss rates for our YouTube scenario with a repofile threshold of 8GB. Because file sizes are larger compared to the webserver scenario, we use larger repofile thresholds for YouTube scenarios than for webserver scenarios. With DR50, the overall hit rate is 95.3% and with DR25 it is 96.9%. The relative differences between DR50 and DR25 are smaller compared to the webserver scenario. This is due to the fact that the probability for requests in most popular files in classes 0 and 1 are larger for a Zipf distribution with $\alpha = 2$ (YouTube) than $\alpha = 1$ (webserver), i.e., 62.7% and 15.7% instead of 27.8% and 13.9%. Therefore, more than 78% of all requests in the YouTube scenarios request content from popularity classes 0 and 1. Since our implementation keeps the most popular files, there is no difference for DR50 and DR25 for most of the requests. However, for class numbers larger than 2, DR25 results in 4.9% to 12% higher hit rates than DR50. We notice a large variability in performance for popularity class numbers larger than 3 in Figure 5b. The variability is much larger than for the webserver scenario in Figure 5a. This can be explained by two reasons. First, the request frequency of class numbers larger than 3 is higher with Zipf distribution $\alpha = 1$ compared to $\alpha = 2$ due to larger request probabilities. Second, the file ranges that we selected in the YouTube scenario are larger than for the webserver scenario resulting in higher variability. When considering the average values, DR25 results in higher hit rates than miss rates up to popularity class 17, while for DR50 the miss rates become higher already at popularity class 11.
Figures 5c and 5d show the hit and miss rates for the webserver scenario with a repofile threshold of 4GB and the YouTube scenario with a repofile threshold of 12GB. Similarly as above, DR25 results in superior performance compared to DR50. In the webserver scenario, DR25 results in

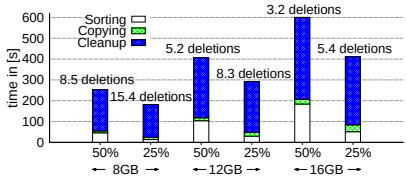
an overall hit ratio of 93%, while for DR50 it is 90.7%. In the YouTube scenario, DR25 results in an overall hit rate of 99% and with DR50 it still reaches 98.1%. Although these values are much higher than in the webserver scenario, miss rates for popularity classes numbers larger than 12 may become larger than the hit rates in the YouTube scenario (worst case) due to a large variability.
Figures 5e and 5f show that if we further increase the repofile threshold to 8GB in the webserver scenario and 16GB in the YouTube scenario, the average hit rates do not go below 80%. Even in the worst case in the YouTube scenario, the hit rates are always higher than miss rates.

4.3 Deletion Times

In this section, we evaluate the time durations to perform deletion operations in the webserver and YouTube scenario.



(a) webserver scenarios



(b) YouTube scenarios

Figure 6: Deletion times and number of deletions

Figure 6 illustrates the overall times for deletions and the number of deletions in each evaluated scenario. The deletion time is split into subparts for sorting segments of deleted content, copying files from the repofile and cleanup of the B-tree. The number on top of each bar denotes the average number of deletions in the corresponding configuration.
Figure 6a illustrates the deletion times as well as the number of deletions in webserver scenarios with repofile thresholds of 2GB, 4GB and 8GB. For every repofile threshold, the percentage on the x-axis denotes deletion ratio of 50% (DR50) and 25% (DR25). For small repofile thresholds of 2GB,

deletions with DR25 take only 16.2% less time than for DR50. This is because most of the time is required for updating file references and deleting entries in the B-tree (cleanup). For DR50, B-tree cleanup requires 86% of the total deletion time and for DR25, it requires even 92%. With increasing repofile threshold, the time for sorting increases. For a repofile threshold of 8GB and DR50, sorting is responsible for 41.5% of the deletion time and for DR25 it is a smaller fraction of 18.6%. DR50 requires more than 300% additional time for sorting and 24.8% more time for cleanup compared to DR25, while their difference for copying is insignificant. For one deletion operation with repofile threshold 8GB, DR25 requires 45% less time for the deletion than DR50. Considering that DR25 requires nearly twice as many deletions over one day, DR25 results only in 6% longer deletion times than DR50. However, due to the increased hit rate for DR25 (see Section 4.2), it may be worth investing 6% more time for deletion.
Figure 6b shows the deletion times for the YouTube scenario. A deletion operation with a repofile threshold of 8GB takes considerably less time than for the webserver scenario: for DR50, it is 36.4% less time and for DR25 16.9% less time. Because files are larger in the YouTube scenario, i.e., have more segments, fewer files are stored in the repository for the same repofile threshold. Due to the sequential request strategy in CCN, segments of the same file are already (more or less) ordered. However, because popular files are continuously pushed down in the *delete_queue*, the sorting overhead increases with the number of deleted files, i.e., no First-In-First-Out (FIFO) deletion strategy. Fewer files that contain more ordered segments (YouTube scenario) result in a less fragmented repository file than many files with fewer ordered segments (webserver scenario) and can, therefore, be sorted faster. As a result, sorting requires only 18.1% for DR50 and only 7.5% for DR25. However, the larger the repository file becomes, the higher is the overhead for sorting and cleanup. For a repofile threshold of 16GB, sorting is responsible for 30.6% of the deletion time for DR50 and 12.5% for DR25. Although sorting takes 3.5 times more time with DR50 compared to DR25, a deletion operation with DR25 only requires 31.6% less time compared to DR50. This is because the overhead from copying is not negligible anymore, i.e., 41.6% more time for DR25, and cleanup becomes more expensive for DR25, i.e., only 16.9% less time compared to DR50. When taking into account the number of deletions, DR25 results in 15.6% longer deletion times. Considering that the overall hit rate for DR50 and DR25 is almost the same (less than 1% difference), it may be a better strategy to use DR50 instead of DR25 in the YouTube scenario.

5 Discussion

5.1 Chunk-based vs. Object-based Persistent Caching

We process content based on object granularity, which may have disadvantages in some cases. For example, if only the first few seconds of a 2 hours movie would be retrieved, the content would be considered as popular as if the entire 2 hours would be requested. However, because content is requested sequentially, i.e., up to *n* segments at the same time depending on the pipeline size, high delay variability between segments would drastically degrade download performance. As a consequence, storing chunks individually may harm download performance.
However, please note that our approach is an extension for persistent storage of static files, but it can be combined with regular caching in the content store, e.g., real-time data, which can be chunk-based. It may also be possible to increase the granularity in the *delete_queue* by splitting each content into ranges of multiple consecutive segments, e.g., segments 0 - 100, 101 - 200, ... and so on, but it would result in a higher processing complexity (tradeoff).
Another challenge of object-based granularity is the update process, i.e., when content needs to be pushed back to the end of the *delete_queue*. In our approach, we assume that the first segment is requested in every download and, therefore, pushes content entries back based on requests in the first segment. We decided to go with this strategy to avoid too many push operations, e.g., for concurrent requests, and to not discriminate small files with only a few segments compared to large files, which would have much more update operations. However, if the first segment is not requested, the content entry is not pushed back. This issue may be alleviated when using higher content object granularity, e.g., ranges of multiple consecutive segments as described above.

5.2 Deletion Overhead

Our evaluations have shown that deletion operations result in a large processing overhead to update the B-tree, sort the deleted segments and delete content from the repofile. For some applications, the processing overhead may not be critical, e.g., delay-tolerant networking [13], [14] or custodian-based information sharing [15], and it can be performed as

maintenance operation during off-peak hours.
However, if persistent caching is used alongside content routers, service interruptions (see Subsection 3.2.3) have a larger impact on caching performance because no new content can be included during deletion operations. In this case, a (read-only) repository could be used during the deletion to continuously serve existing content. In addition, a router may use multiple repositories at the same time (load balancing). Then, if one repository performs a deletion, the other repository may still accept content. In the worst case, i.e., if only one repository is used and Interests cannot be satisfied due to a local deletion, Interests may be forwarded to the next content router on the path to the content source.

6 Conclusions

Persistent caching is required to support delay-tolerant networking or provide high content availability similar to content delivery networks (CDNs). In this work, we have extended the current repository implementation in CCNx to support persistent caching in repositories. A fundamental requirement for persistent caching is content deletion during operation, i.e., without deleting or resetting the entire repository, which is not supported by CCNx. Our approach for content deletion is based on a *delete_queue*, which keeps the most popular files at the tail of the queue. If disk space needs to be released, content can be removed from the head of the queue.

We have performed extensive experimental evaluations for different configurations in a webserver and YouTube scenario. In every scenario, new content has been generated periodically such that deletion operations in the repository were necessary due to limited space. Evaluations have shown that our design can maintain high cache hit rates in both scenarios, but performance depends on the reserved repofile size for caching. Although repositories are slightly larger in the YouTube scenario due to larger file sizes, the repositories need to store a smaller percentage of all content to achieve high cache hit rates. For example, in webserver scenarios a repofile size of 4GB, which corresponds to 21% of all included content during a day, results in cache hit rates larger than 90%. In YouTube scenarios, a repofile size of 12GB, which corresponds to 14% of all included content in a day, results in cache hit rates larger than 98%. High cache hit rates at the edge are beneficial for both users and network operators. While network operators can reduce network traffic at the core network to improve network availability and reduce operational costs, users may benefit from faster content downloads (shorter delays, less RTT variability) as well as partial service and content availability if the core network is overloaded.

In the webserver scenario, it is a better strategy to have more frequent deletions of fewer content to obtain higher cache hit rates. More frequent deletions of fewer files do not require much more time for deletions, i.e., longer service interruptions, than fewer deletions of many files. In the YouTube scenario, fewer but larger deletions are better. Compared to the webserver scenario, the sorting overhead is significantly smaller because fewer files can be stored at the same space (files are larger) and segments of the same file are already ordered. In addition, because most requests are addressed to the most popular classes, the additional gain of keeping less popular content in the repository is only minimal and may not justify more frequent deletion operations.

We have implemented persistent caching based on the current repository implementation in CCNx, which uses a B-tree to keep references to stored content in the filesystem. For caching in delay-tolerant networking, the overhead for deletions may be negligible but to increase efficiency in content routers, other repository implementations may be evaluated, e.g., storing files or even chunks in separate repofiles to reduce cleanup and sorting overhead, or even develop a repository implementation with a database. However, all of these solutions would come with their own disadvantages that would need to be evaluated.

References

[1] D.-h. Kim, J.-h. Kim, Y.-s. Kim, H.-s. Yoon, and I. Yeom, "Mobility Support in Content Centric Networks," in *Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking*, ser. ICN '12. New York, NY, USA: ACM, 2012, pp. 13–18.

[2] D. Smetters and V. Jacobson, "Securing network content," PARC, Tech. Rep., Oct. 2009. [Online]. Available: https://www.parc.com/content/attachments/TR-2009-01.pdf

[3] I. Psaras, R. Clegg, R. Landa, W. Chai, and G. Pavlou, "Modelling and evaluation of CCN-caching trees," in *Proceedings of the 10th international IFIP TC 6 conference on Networking*, Valencia, Spain, May 2011, pp. 78–91.

[4] I. Psaras, W. Chai, and G. Pavlou, "Probabilistic In-Network Caching for Information-Centric Networks," in *Proceedings of the 2nd ACM SIGCOMM workshop on Information-centric networking (ICN)*, Helsinki, Finland, August 2012, pp. 50–60.

[5] S. Eum, K. Nakauchi, M. Murata, Y. Shoji, and N. Nishinaga, "CATT: potential based routing with content caching for ICN," in *Proceedings of the 2nd ACM SIGCOMM workshop on Information-centric networking (ICN)*, Helsinki, Finland, August 2012, pp. 49–54.

[6] Y. Li, T. Lin, H. Tang, and P. Sun, "A Chunk Caching Location and Searching Scheme in Content Centric Networking," in *IEEE International Conference on Communications (ICC)*, Ottawa, Canada, June 2012, pp. 2655–2659.

[7] K. Cho, M. Lee, K. Park, T. Kwon, Y. Choi, and S. Pack, "WAVE: Popularity-based and collaborative in-networking caching for content-oriented networks," in *IEEE INFOCOM workshop on Emerging Design Choices in Name-Oriented Networking (NOMEN)*, Orlando, FL, USA, March 2012, pp. 316–321.

[8] Q. Wu and Z. L. an G. Xie, "CodingCache: Multipath-aware CCN Cache with Network Coding," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking (ICN)*, Hong Kong, China, August 2013, pp. 41–42.

[9] J. Wang, J. Ren, K. Lu, J. Wang, S. Liu, and C. Westphal, "An Optimal Cache Management Framework for Information-Centric Networks with Network Coding," in *Proceedings of the 13th IFIP Networking Conference*, Trondheim, Norway, June 2014, pp. 1–9.

[10] Cisco, "Visual Networking Index (VNI): Global Mobile Data Traffic Forecast Update, 2014 - 2019," February 2015.

[11] A. Gomes and T. Braun, "Feasibility of Information-Centric Networking Integration into LTE Mobile Networks," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, Salamanca, Spain, April 2015.

[12] D. Perino and M. Varvello, "A Reality Check for Content Centric Networking," in *Proceedings of the ACM SIGCOMM workshop on information-centric networking (ICN)*, Toronto, Canada, August 2011, pp. 44–49.

[13] C. Anastasiades, T. Schmid, J. Weber, and T. Braun, "Opportunistic Content-Centric Data Transmission During Short Network Contacts," in *IEEE Wireless Communications and Networking Conference (WCNC)*, Istanbul, Turkey, April 2014, pp. 2516–2521.

[14] C. Anastasiades, W. El Maudni El Alami, and T. Braun, "Agent-based Content Retrieval for Opportunistic Content-Centric Networks," in *12th International Conference on Wired & Wireless Internet Communications (WWIC)*, ser. LNCS 8458, Paris, France, May 2014, pp. 175–188.

[15] V. Jacobson, R. L. Braynard, T. Diebert, P. Mahadevan, M. Mosko, N. Briggs, S. Barber, M. Plass, I. Solis, E. Uzun, B. Lee, M.-W. Jang, D. Byun, D. K. Smetters, and J. D. Thornton, "Custodian-based information sharing," *IEEE Communications Magazine*, vol. 50, no. 7, pp. p. 38–43, July 2012.

[16] H. Sarkissian, "The Business Case for Caching in 4G LTE Networks," http://www.wireless2020.com/docs/LSI_WP_Content_Cach_Cv3.pdf, LSI Corporation, April 2013.

[17] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Network Named Content," in *Proceedings of the 5th ACM international conference on Emerging networking experiments and technologies (CoNEXT)*, Rome, Italy, December 2009, pp. 1–12.

[18] CCNx. (2015, March) http://www.ccnx.org/.

[19] U. Lee, I. Rimac, and V. Hilt, “Greening the Internet with Content-Centric Networking,” in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking (e-Energy)*, Passau, Germany, April 2010, pp. 179–182.

[20] UBELIX. University of Bern Linux Cluster. [Online]. Available: http://www.ubelix.unibe.ch

[21] D. Rossi and G. Rossini, “Caching performance of content centric networks under multi-path routing (and more),” Telecom ParisTech, Tech. Rep., 2011.

[22] T. Yu, C. Tian, H. Jiang, and W. Liu, “Measurements and Analysis of an Unconstrained User Generated Content System,” in *IEEE International Conference on Communications (ICC)*, Kyoto, Japan, June 2011, pp. 1–5.

[23] TubeMogul. (2009, May) Half of youtube videos get fewer than 500 views. [Online]. Available: http://www.businessinsider.com/chart-of-the-day-youtube-videos-by-views-2009-5?IR=T

[24] A. Abhari and M. Soraya, “Workload generation for youtube,” *Multimedia Tools and Applications*, vol. 46, no. 1, pp. 91–118, January 2010.

[25] A. Williams, M. Arlitt, C. Williamson, and K. Baker, “Web workload characterization: Ten years later,” *Web Content Delivery*, vol. 2, no. 1, pp. 3–21, 2005.